A Consistent History Link Connectivity Protocol *

Paul LeMahieu Jehoshua Bruck

California Institute of Technology
Mail Code: 136-93
Pasadena, CA 91125

Email: {lemahieu,bruck}@paradise.caltech.edu

Abstract

The RAIN (Reliable Array of Independent Nodes) project at Caltech is focusing on creating highly reliable distributed systems by leveraging commercially available personal computers, workstations and interconnect technologies. In particular, the issue of reliable communication is addressed by introducing redundancy in the form of multiple network interfaces per compute node.

When using compute nodes with multiple network connections the question of how to determine connectivity between nodes arises. We examine a connectivity protocol that guarantees that each side of a point-to-point connection sees the same history of activity over the communication channel. In other words, we maintain a *consistent history* of the state of the communication channel. At any give moment in time the histories as seen by each side are guaranteed to be identical to within some number of transitions. This bound on how much one side may lead or lag the other is the *slack*.

Our main contributions are: (i) a simple, *stable* protocol for monitoring connectivity that maintains a *consistent history* with *bounded slack*, and (ii) proofs that this protocol exhibits *correctness*, *bounded slack*, and *stability*.

*Supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, and by DARPA through an agreement with NASA/OSAT.

1 Introduction

Given the prevalence of powerful personal workstations connected over local area networks, it is only natural that people are exploring distributed computing over such systems. Whenever systems become distributed the issue of fault tolerance becomes an important consideration. In the context of the *RAIN* project (Redundant Arrays of Independent Nodes) at Caltech (see Figure 14 for a photo), we've been looking into fault tolerance in several elements of the distributed system. One important aspect of this is the introduction of fault tolerance into the communication system by introducing redundant network elements and redundant network interfaces at each compute node. For example, a practical and inexpensive real-world system could be as simple as two Ethernet interfaces per machine and two Ethernet hubs. The work we have done is not specific to any networking technology, but we have been working primarily with Myrinet [3] networking elements as well as with Ethernet networks.

An elementary piece of information about the system is whether there is *connectivity* between an interface on one machine and an interface on another. We describe here a modified *ping* protocol that guarantees that each side of the communication channel sees the same history. Each side is limited in how much it may lead or lag the other side of the channel, giving the protocol *bounded slack*. This notion of identical history can be useful in the development of applications using this connectivity information. For example, if an application takes error recovery action in the event of lost connectivity, it knows that both sides of the channel will see the exact same behavior on the channel over time, and will thus take the same error recovery action. Such a guarantee may simplify the writing of applications using this connectivity information.

The protocol can be run at a high level, potentially as high as the application layer. At the same time, it can benefit greatly from low-level hardware hints about link conditions. A primary application for such a protocol is within a communication layer where its information can be used to mask or report connectivity problems between machines.

Although in some sense this is a consensus problem since the history of channel activity must be seen the same at both sides, but it is not the general consensus problem people think of. We are only really interested in *eventual* consensus when the link is functioning. When the link has failed, we only care that the nodes see the failure. However, it is still useful to look at past work on consensus, such as Fischer, Lynch, and Paterson in [6], or in Lynch's book [8].

The connectivity problem has been addressed with different goals by Rodeheffer and Schroeder in the Autonet system [9, 10]. They were concerned with adaptive rates and skepticism in judging the quality of a link, whereas we are concerned with consistency in reporting the quality of a link. The connectivity problem has no doubt been considered in routing algorithms in the past, but we have seen no reference to keeping the history consistent at each side of a link.

Other than our own practical motivation for a consistent history of the channel state, Birman [1] gives general motivation for consistency in failure reporting for the purpose of improving reliability of distributed systems.

Our main contributions are: (i) a simple, *stable* protocol for monitoring connectivity that maintains a *consistent history* with *bounded slack*, and (ii) proofs that this protocol exhibits *correctness*, *bounded slack*, and *stability*.

The structure of the paper follows closely the contributions listed above. In Section 2 we define the problem. In Section 3 we explain the protocol. In Section 4 we prove the protocol satisfies all requirements. In Section 5 we discuss implementation issues in our existing system, and in 6 we finish with conclusions.

2 Problem Definition

We consider the following problem: given two nodes connected by some bidirectional communication channel, what is the state of the channel connecting them?

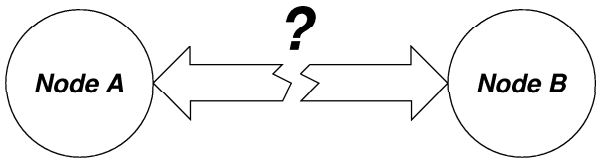

Figure 1: Node A, node B, and the communication channel between them. Is the channel *Up* or *Down*?

We desire a stable protocol that guarantees both sides see the same history of the channel up to some given *bounded slack*. Figure 2 shows what we desire in a consistent history between ends of a communication channel. We desire that neither side be permitted to lag or lead the other by an arbitrary number of observed channel state transitions.

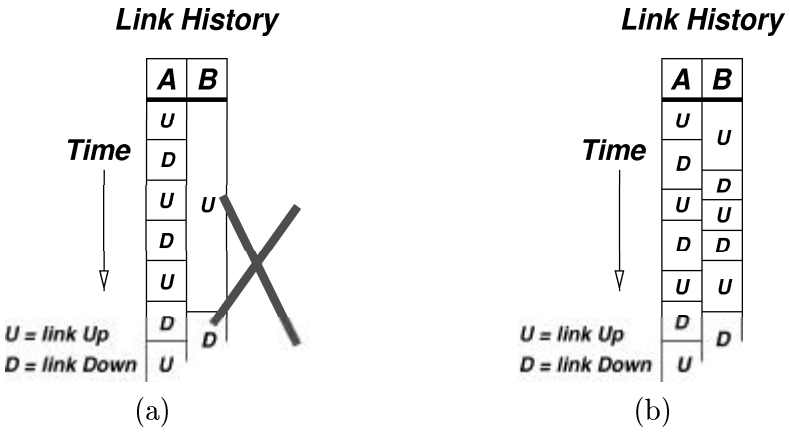

Figure 2: (a) An invalid history, with node A seeing many more transitions in the channel state than node B. (b) A valid history, with both node A and B showing tightly coupled views of the channel state.

2.1 Defined Terminology

Below, we define some of the terminology used in this paper.

Nodes and Communication Channels. A distributed computing system is composed of a set of interconnected *nodes*. We are unconcerned with the underlying interconnect, but are interested in the existence of logical *communication channels* between a node and the other nodes in the system, on a point-to-point basis. The protocol runs over a pair of nodes connected by a communication channel.

Connectivity and Channel State. We consider two compute nodes *connected* only if bidirectional communication exists between them. Bidirectional communication is necessary for the implementation of reliable communication over unreliable channels. If a node finds itself connected to another node via a given channel, it considers that channel in the *Up* state. If a node finds itself not connected to another node via a given channel, it considers that channel in the *Down* state.

History. The sum of decisions made up about the state of a channel makes up that channel's *history*. Each of the endpoints of the channel adds to its view of the channel-state history each time it decides the channel is *Up* or *Down*. A channel's history will be a series of channel states: *Up*, *Down*, *Up*, *Down*,.... Since the channel state is binary, a simple count of the number of state transitions suffices to fully describe the history.

Slack. As a node makes decision about the state of a channel, it may lead or lag the node on the opposite end of the channel. *Slack* is the amount a node may lead or lag its peer node. If t_a and t_b are the number of channel state transitions seen by node A and node B, respectively, and N is the slack parameter, than $|t_a - t_b| \leq N$ at all times.

Real Channel Event. A *real channel event* would be any spontaneously occurring information about the channel. The simplest would be "the channel appears to be up" or "the channel appears to be down." We'll look at the *timeout* event that signifies that bi-directional communication has been lost. We'll also permit the *timein* event, the complement to the timeout, that signifies that bi-directionally communication has been re-established. These are *real* events in the sense that they reflect information about channel activity beyond our control, not an event due to the protocol itself.

Stability. A protocol determining the state of a communication channel should be *stable*. More precisely, for each channel event some bounded number of transitions (preferably one) should be seen by each endpoint.

Reliable Message Passing. The protocol we will describe requires *reliable message passing*. Since this is a protocol intended to work over unreliable channels, we are referring to software implemented reliability, such as some sort of sliding window protocol. We require message passing that gives (eventual) guaranteed, in-order delivery.

2.2 Precise Problem Definition

We now present all the requirements of the protocol:

- *Correctness*: the protocol will eventually correctly reflect the true state of the channel. If the channel ceases to perform bi-directional communication (at least one side sees timeouts), both sides should eventually mark the channel as *Down*. If the channel resumes bi-directional communication, both sides should eventually mark the channel as *Up*.
- *Bounded Slack*: the protocol will ensure a maximum slack of N exists between the two sides. Neither side will be allowed to lag or lead the other by more than N transitions.
- *Stability*: each real channel event (i.e., timeout) will cause at most some bounded number of observable state transitions, preferably one, at each endpoint.

The system model is one in which nodes do not fail, but links intermittently fail. The links must be such that a sliding window protocol can function. See the discussion on data link protocols by Lynch in [8].

3 The Link Connectivity Protocol

This protocol uses *reliable message passing* to ensure that nodes on opposing ends of some faulty channel see the same state history of link failure and recovery. The reliable message passing can be implemented using a sliding window protocol, as mentioned above. At first it may seem odd to discuss monitoring the status of a link using reliable messages. However, it makes the description

and proof of the protocol easier, preventing us from essentially re-proving sliding window protocols in a different form. For actual implementation, there is no reason to actually build the protocol on an existing reliable communication layer. The protocol can be easily implemented on top of ping messages (sent unreliably) with only a sequence number and acknowledge number as data (in other words, we can easily map reliable messaging on top of the ping messages).

The protocol consists of two parts:

- First, we have the sending and receiving of tokens using reliable messaging. Tokens are conserved, neither lost nor duplicated. Tokens are sent whenever a side sees an observable channel state transition. The observable channel state is whether the link is seen as *Up* or *Down*. The token-passing part of the protocol essentially *is* the protocol. It's job is to ensure that a consistent history is maintained.

- Second, we have the sending and receiving of ping messages using unreliable messaging. The sole purpose of the pings is to detect when the link can be considered *Up* or *Down*. This part of the protocol would not necessarily have to be implemented with pings, but could be done using other hints from the underlying system. For example, hardware could give instant feedback about its view of link status. For all the proofs to be valid, we must have that a t_{out} is generated when bi-directional communication has (probably) been lost, and a t_{in} is generated when bi-directional communication has (probably) been re-established.

The token-passing part of the protocol maintains the consistent history between the sides, and the pings give information on the current channel state. The token-passing protocol can be seen as a filter that takes raw information about the channel and produces channel information guaranteed to be (eventually) consistent at both ends of the channel. The state machines in Figure 3 and Figure 4 describe how each side of the protocol functions in the total system.

In Section 3.1 we describe the protocol for a slack of $N = 2$, and in 3.2 we do so for a general slack of N. In Sections 4.1, 4.2, and 4.3 we establish correctness, bounded slack, and stability for the protocol.

3.1 Slack $N = 2$

Here we describe the protocol for the base case where we have slack of $n = 2$. This is a significant case since it is the smallest value of slack for which any such protocol can work. It's description is hopefully somewhat simpler than the general case. A little state machine as described in Figure 3 runs at each end of the link, at each node.

3.2 General Slack N

Here we describe the protocol for the general case where we have some arbitrary slack value, N. A state machine as described in Figure 4 runs at each end of the communication, at each node. This description of the state machine tries to preserve the same structure as the $N = 2$ machine described above for the purpose of the proofs. It looks more complicated than it really is. See Appendix A for an equivalent 2-state machine parameterized by the token count t that is more appropriate when it comes to implementing the protocol.

Notice that here we've introduced the t_{in} event, which was not present in the $N = 2$ case. This is the complement to the t_{out} event that becomes meaningful for higher slack. A t_{in} corresponds to a hint from some lower level that the communication link is up and running. For an implementation where tokens are mapped on top of pings, we would never explicitly see a t_{in} event since the latest

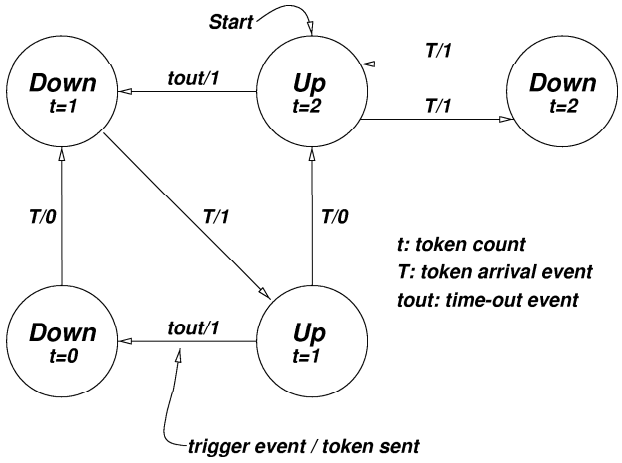

Figure 3: State machine for the connectivity protocol, slack $N = 2$. Each state is characterized by whether the node sees the channel as *Up* or *Down*, and how many tokens t are held by the node. The state transitions are labeled by the *action triggering the transition*, and the *action taken upon transition*. A trigger event is either a timeout t_{out} or receipt of a token T. The action taken is always whether a token is sent (1) or not (0).

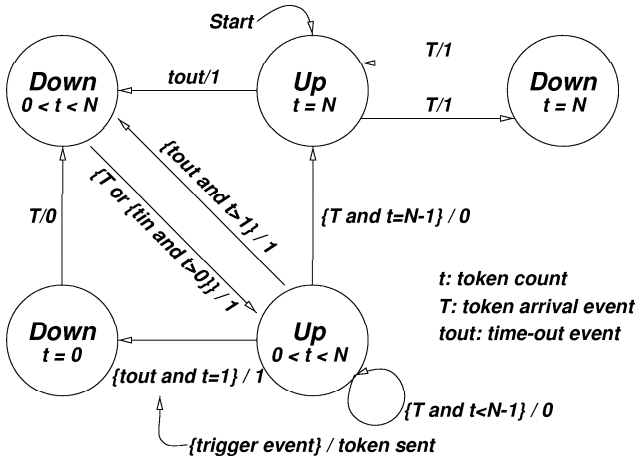

Figure 4: State machine for the connectivity protocol, general slack N. Each state is characterized by whether the node sees the channel as *Up* or *Down*, and how many tokens t are held by the node. There is an implicit action for each transition: if a token is received the token count t is incremented; if a token is sent the token count t is decremented. The state transitions are labeled by the pair $\{$*action triggering the transition*$\}$ / $\{$*action taken upon transition*$\}$. A trigger event is either a timeout t_{out}, a time-in t_{in}, or receipt of a token T. The action taken is a combinations of sending tokens and adjusting the token count t.

token information comes with each ping response. However, if tokens were not mapped on top of pings, or if other sources of information about the communication link were also possible, t_{in} events make sense and as such are allowed.

4 Proofs of Protocol Properties

In the following three subsections we establish that the protocol exhibits bounded slack, stability, and correctness.

4.1 Bounded Slack

This theorem and proof is not actually specific to the protocols given above. A limited subset of the protocols (the token passing conditions) are sufficient to establish that slack is bounded.

Theorem 1 We take any protocol between two communicating nodes (A and B) with the following characteristics:

1. Each side starts with N tokens.
2. Tokens are never generated or destroyed
3. Tokens are sent exactly when a node decides the channel has made a change of state ($Up \rightarrow Down$ or $Down \rightarrow Up$). In other words, tokens are sent for *observable state transitions* of the node.

Any protocol that meets these criteria will have a *bounded slack* property. If we call d_A and d_B the observable state transitions for node A and B, respectively, then

$$|d_A - d_B| \leq N$$

Proof:
Define the following values:

$$\begin{aligned} d_A,\ d_B &= \text{number of tokens sent by node A (node B)} \\ t_A,\ t_B &= \text{number of tokens held at node A (node B)} \\ t_{AB},\ t_{BA} &= \text{number of tokens in the channel from A to B (B to A)} \\ N_A &= t_A + t_{BA} \\ N_B &= t_B + t_{AB} \end{aligned}$$

The proof will examine one of the two regions shown in Figure 5, in particular, the region made up of the channel from A to B, and of node B itself. By simply counting the tokens originally present in the region and those that enter and leave the region, we will establish that slack is bounded between the two sides.

By simple conservation of tokens, it is invariantly true that

$$N_B = \underbrace{\text{tokens initially present}}_{N} + \underbrace{\text{tokens that have entered}}_{d_A} - \underbrace{\text{tokens that have exited}}_{d_B}$$

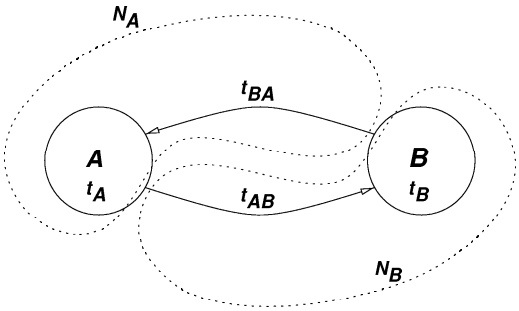

Figure 5: Node-Channel system partitioned into two halves.

So,

$$t_B + t_{AB} = N + d_A - d_B \tag{1}$$

We do the proof by contradiction. Suppose the slack bound is violated, $d_A - d_B > N$. Then, substituting into (1),

$$t_B + t_{AB} > 2N \tag{2}$$

This is our contradiction, since we have only $2N$ tokens in the system. □

4.2 Stability

By *stability*, we mean that the protocol will exhibit finite response to a physical (timeout) event. We wouldn't want a protocol that could repeatedly mark a channel as *Up* and *Down* in response to a single timeout. We require that the number of channel-state transitions is bounded by the number of physical timeouts in the system. More specifically, we will show that every timeout causes at most two $U \rightarrow D$ transitions: one at the side that sees the timeout explicitly, and possibly one at the peer node. We'll first present the proof for the slack 2 case, and then present the general slack-N proof.

4.2.1 Slack $N = 2$

Theorem 2 For a system comprised of two nodes each running the slack-2 state machine of Figure 3 and connected by a bi-directional communication channel, every *Up* $\rightarrow$ *Down* transition is directly caused by a timeout.

Proof:
First, we label the tokens in the state machine of Figure 3, transforming it to the state machine of Figure 6. We have labeled the tokens as follows:

1. A token is labeled D if it was sent for a $U \rightarrow D$ transition.
2. A token is labeled U if it was sent for a $D \rightarrow U$ transition.

Since tokens are always sent D, U, D, U, ..., we can also easily determine what kind of token we will be receiving in a given state. Starting in the initial state, the first token received must be a D token, the next a U token, and so forth. There is a unique type of received token at each state since all cycles involve an even number of received tokens.

Let's look at a system made of Node A and Node B, each running such a state machine. As stated above, it is sufficient to concern ourselves with the $U \rightarrow D$ transitions made by a given side.

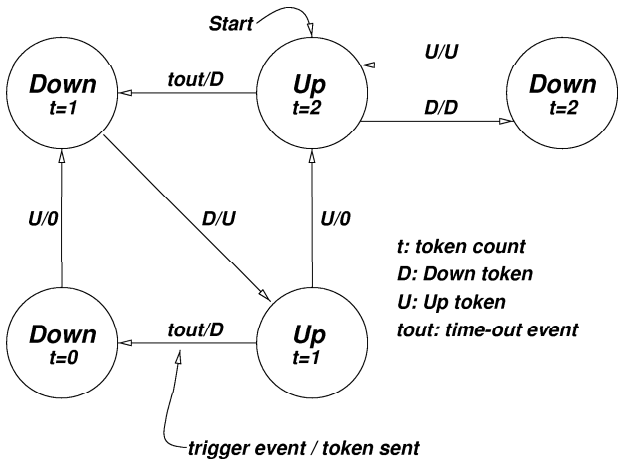

Figure 6: Token-labeled state machine for the connectivity protocol, slack $N = 2$.

Assume A is making such a transition. We need to show that we have each $U \to D$ transition is either caused by a local t_{out}, or is the direct result of a t_{out} at Node B.

Node A: $U \to D$ transitions	t_{out} accounted for?
$t_{out} \Rightarrow A_{U_2 \to D_1}$	*yes*
$t_{out} \Rightarrow A_{U_1 \to D_0}$	*yes*
D-tok $\Rightarrow A_{U_2 \to D_2}$	*no*

(a)

Node B: transitions generating D-tok	t_{out} accounted for?
$t_{out} \Rightarrow B_{U_2 \to D_1} \Rightarrow$ D-tok	*yes*
$t_{out} \Rightarrow B_{U_1 \to D_0} \Rightarrow$ D-tok	*yes*
D-tok $\Rightarrow B_{U_2 \to D_2} \Rightarrow$ D-tok	*no*

(b)

Figure 7: (a) Node A: $U \to D$ transitions (b) Node B: transitions generating D-tok (i.e., all $U \to D$ transitions)

At Node A, we have such transitions resulting from timeouts, and one such transition resulting from receipt of a D-token (see Figure 7a). The first two cases are simply $U \to D$ transitions due to local timeouts, and need no further examination. The third case needs to be explored in more detail by looking at where the D-token could have come from. At Node B, a D-token could have been generated by a $U \to D$ transition caused by a timeout, or generated by a $U \to D$ transition caused by receipt of a D-token (see Figure 7b). Again, the first two cases are simply $U \to D$ transitions due to local timeouts, and need no further examination. The third case is where the proof is needed. We have this problem: can the third cases of the two lists above occur in a chain? I.e., we wonder whether the following sequence can occur:

$$\text{D-tok} \Rightarrow B_{U_2 \to D_2} \Rightarrow \text{D-tok} \Rightarrow A_{U_2 \to D_2}$$

This is the only case where we have a chain of events involving no timeouts that has $U \rightarrow D$ transitions in it. If we prove this can never happen, we are done. We prove this by contradiction.

Much as in the previous proof, we'll separate the system into two separate components and use a token-counting argument to show this case is impossible. Even at a glance the problem is obvious since we have a state with two tokens sending a token to another state with two tokens, yet we only have 4 total tokens in the system. The asynchronous nature of these interactions is the only thing stopping us from already being done with the proof.

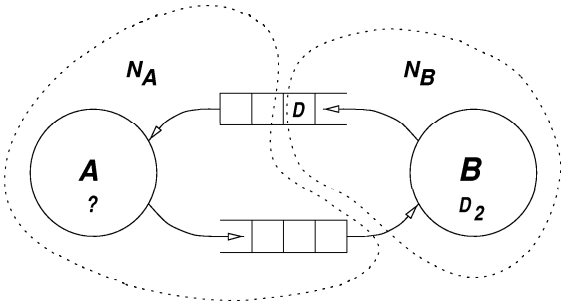

Figure 8: Partitioning the system: Node B is in state D_2 and has sent a D-tok to Node A; we make no claim on Node A's state.

First, we look at the system at the point where Node B has just sent the D-token to Node A (see Figure 8). Here you see how we've partitioned the system: we include in one partition all those tokens at Node B, plus all those in the FIFO up to and including the D-token just sent. We call this number of tokens N_B, and we see that $N_B \geq 3$ initially since Node B is in state D_2 and there is the D-token in the channel. The rest of the system makes up the second partition, and has N_A tokens. Initially no claim is made as to the size of N_A.

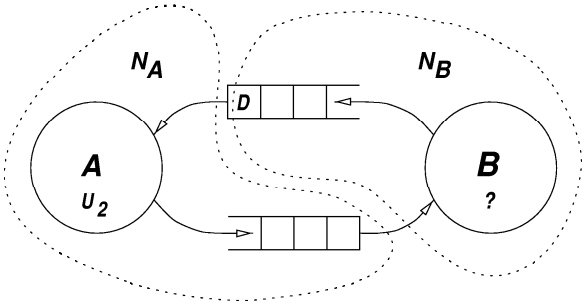

Figure 9: Partitioning the system: Node A is in state U_2 and is about to receive the D-tok from Node B; we make no claim on Node B's state.

Second, we look at the system at the point where Node A is just about to receive the D-token sent by Node B (see Figure 9). The partitioning is the same, but time has advanced. Now we see that $N_A \geq 2$ since Node A is in state U_2. However, $N_B \geq 3$ still holds since no tokens have left that component. This is our contradiction since there are only 4 tokens in the system. □

4.2.2 General Slack N

Theorem 3 For a system comprised of two nodes each running the general slack-N state machine of Figure 4 and connected by a bi-directional communication channel, every $Up \rightarrow Down$ transition is directly caused by a timeout.

Proof:
First, we label the tokens in the state machine of Figure 4, transforming it to the state machine of Figure 10. We have labeled the tokens as follows:

1. A token is labeled D if it was sent for a $U \rightarrow D$ transition.
2. A token is labeled U if it was sent for a $D \rightarrow U$ transition.

Since tokens are always sent D, U, D, U, ..., we can also easily determine what kind of token we will be receiving in certain states. Starting in the initial state, the first token received must be a D token, the next a U token, and so forth.

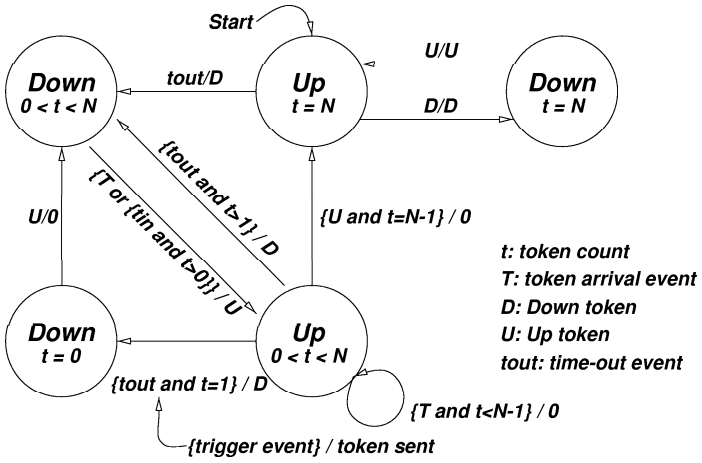

Figure 10: Token-labeled state machine for the connectivity protocol, general slack N.

Let's look at a system made of Node A and Node B, each running such a state machine. As stated above, it is sufficient to concern ourselves with the $U \rightarrow D$ transitions made by a given side. Assume A is making such a transition. We need to show that we have each $U \rightarrow D$ transition is either caused by a local t_{out}, or is the direct result of a t_{out} at Node B.

At Node A, we have such transitions resulting from timeouts, and one such transition resulting from receipt of a D-token (see Figure 11a). The first three cases are simply $U \rightarrow D$ transitions due to local timeouts, and need no further examination. The fourth case needs to be explored in more detail by looking at where the D-token could have come from. At Node B, a D-token could have been generated by a $U \rightarrow D$ transition caused by a timeout, or generated by a $U \rightarrow D$ transition caused by receipt of a D-token (see Figure 11b). Again, the first three cases are simply $U \rightarrow D$ transitions due to local timeouts, and need no further examination. The fourth case is where the proof is needed. We have this problem: can the fourth cases of the two lists above occur in a chain? I.e., we wonder whether the following sequence can occur:

$$\text{D-tok} \Rightarrow B_{U_N \rightarrow D_N} \Rightarrow \text{D-tok} \Rightarrow A_{U_N \rightarrow D_N}$$

Node A: $U \to D$ transitions	t_{out} accounted for?
$t_{out} \Rightarrow A_{U_N \to D_{(0<t<N)}}$	*yes*
$t_{out} \Rightarrow A_{U_{(0<t<N)} \to D_0}$	*yes*
$t_{out} \Rightarrow A_{U_{(0<t<N)} \to D_{(0<t<N)}}$	*yes*
D-tok $\Rightarrow A_{U_N \to D_N}$	*no*

(a)

Node B: transitions generating D-tok	t_{out} accounted for?
$t_{out} \Rightarrow B_{U_N \to D_{(0<t<N)}} \Rightarrow$ D-tok	*yes*
$t_{out} \Rightarrow B_{U_{(0<t<N)} \to D_0} \Rightarrow$ D-tok	*yes*
$t_{out} \Rightarrow B_{U_{(0<t<N)} \to D_{(0<t<N)}} \Rightarrow$ D-tok	*yes*
D-tok $\Rightarrow B_{U_N \to D_N} \Rightarrow$ D-tok	*no*

(b)

Figure 11: (a) Node A: $U \to D$ transitions (b) Node B: transitions generating D-tok (i.e., all $U \to D$ transitions)

This is the only case where we have a chain of events involving no timeouts that has $U \to D$ transitions in it. If we prove this can never happen, we are done. We prove this by contradiction.

Much as in the previous proof, we'll separate the system into two separate components and use a token-counting argument to show this case is impossible. Even at a glance the problem is obvious since we have a state with N tokens sending a token to another state with N tokens, yet we only have $2N$ total tokens in the system. The asynchronous nature of these interactions is the only thing stopping us from already being done with the proof.

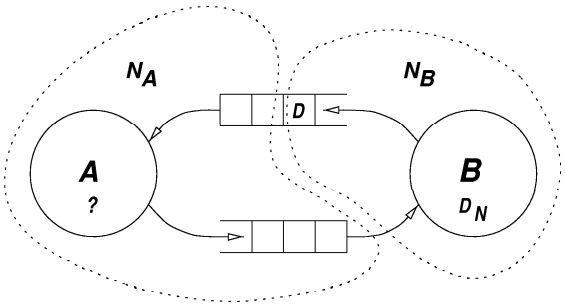

Figure 12: Partitioning the system: Node B is in state D_N and has sent a D-tok to Node A; we make no claim on Node A's state.

First, we look at the system at the point where Node B has just sent the D-token to Node A (see Figure 12). Here you see how we've partitioned the system: we include in one partition all those tokens at Node B, plus all those in the FIFO up to and including the D-token just sent. We call this number of tokens N_B, and we see that $N_B \geq N + 1$ initially since Node B is in state D_N and there is the D-token in the channel. The rest of the system makes up the second partition, and has N_A tokens. Initially no claim is made as to the size of N_A.

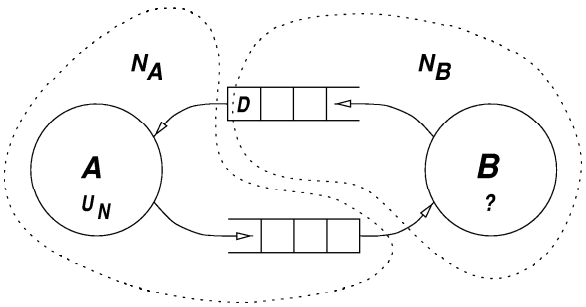

Figure 13: Partitioning the system: Node A is in state U_N and is about to receive the D-tok from Node B; we make no claim on Node B's state.

Second, we look at the system at the point where Node A is just about to receive the D-token sent by Node B (see Figure 13). The partitioning is the same, but time has advanced. Now we see that $N_A \geq N$ since Node A is in state U_N. However, $N_B \geq N + 1$ still holds since no tokens have left that component. This is our contradiction since there are only $2N$ tokens in the system. □

4.3 Correctness

By "correctness" we mean the simplest aspect of correct behavior: (1) the protocol will eventually mark the channel as *Up*, given that neither side sees timeouts (bi-directional communication exists), and (2) the protocol will eventually mark the channel as *Down*, given that both sides see timeouts (bi-directional communication does not exist). This requirement eliminates trivial protocols that always mark the channel as either *Up* or *Down*, and protocols that would allow themselves to get in a state where they must keep reporting the channel as *Up*, despite persistent timeouts, to satisfy the bounded slack property.

1. Bi-directional communication exists: neither side sees timeouts. In a stable system where full communication exists, the channels must eventually be empty. Thus, to account for all tokens each side must have exactly N tokens (the most each side can hold). We must show that the only possible state of the system is for both sides to be in U_N.

   - Consider the case where Node A is in U_N and Node B is in D_N. Looking at the labeled state machine of Figure 10, this would mean that the last token sent by Node B was a D-token and the last token received by Node A was a U-token, which is a contradiction.
   - Consider the case where Node A is in D_N and Node B is in D_N. Both cannot come to rest in D_N states since the transition to this states requires the sending of a token. Say it is Node A which first comes to rest in the D_N state. The D-token it sends out upon making that transition would have to be the one to cause Node B to transition to the D_N state. However, this chain of events is prohibited by the stability argument made in the previous section. Note that not only can the two sides not both stabilize in the D_N state, but it is not even possible for both to be in such a state as a transient condition.

   So, the only allowable stable state is both in U_N. □

2. Bi-directional communication does not exist: both sides see timeouts. This is a simple case, since both sides will see timeouts and transition to a *Down* state if they are not already in one. Note that for this to be true there must always be a transition to a *Down* state from every *Up* state triggered solely by a timeout event. This is true for our state machine. □

One interesting thing to note here is that we cannot prove correctness as specified in Case 2 above if the timeouts seen by the state machine are only generated by loss of one direction of communication. In other words, if it is possible that only one of the nodes continually sees timeouts while the other node sees none, then we cannot guarantee that both sides eventually transition to the *Down* state. This is why we use *pings* to monitor link connectivity rather than each side simply generating a *heartbeat*: we require that timeout events are generated whenever *bi-directional* communication is lost.

5 Implementation and System Notes

Our need for monitoring connectivity arose in the *RAIN* system here at Caltech. *RAIN* (Redundant Arrays of Independent Nodes) is a local network of inexpensive, off-the-shelf computing nodes (see Figure 14). The system was created to test ideas on introducing fault-tolerance into networks of computers. This includes fault-tolerance in distributed storage across the machines as well as fault-tolerance in network connectivity, where the connectivity protocol plays a role.

Figure 14: The *RAIN* System. This is a ten node system. There are ten computers down below, ten screens up on the rack, and Myrinet network switches in the center. They are running a video demo as well as the basic system monitoring software.

Initially, the simplest possible approach was used: simple heartbeats were used to get a current view of the channels. In the implementation of the communication layer it became clear that a more sophisticated reporting of connectivity could simplify the job. In particular, reacting to

(a) (b)

Figure 15: Here are two screen shots. Again we see the video demo and the system monitoring software for a ten node system with two network interfaces per node. (a) Example screen shot showing local view and full connectivity: two lines connect to each other machine in the network. (b) Example screen shot showing local view and detection of failure of some links: loss of one link to each other machine (caused by removing power from one of the switches) and loss of both links to two other machines (caused by disconnecting those two machines from the remaining switch).

Down connections was like taking action for an error condition. If connectivity information could be provided that was identical within a slack of $N = 2$ at both sides of the channel, the communication layer could make the simplifying assumption that both sides would see all the same errors, and that the periods where each side saw the channel *Down* would overlap.

In the absence of a connectivity protocol that provided a bounded slack and consistent history, we saw ourselves essentially implementing the functionality of the protocol we have described, but integrated into the rest of the communication layer. In this way it was harder to analyze and more difficult to argue correctness. We felt the isolation of this protocol from the rest of the implementation served our purposed in establishing correctness of the system, and may help others since it now exists as a proven module with strict specifications. It serves as an excellent exercise in full protocol specification and proof, and gives motivation for new protocols that may ease the solution of different problems by trying to achieve "consistent history" solutions.

As a final implementation note, we would like to stress again that the development of this protocol using reliable communication primitives is merely a tool for specifying and proving the protocol, saving ourselves from essentially reproving sliding window protocols. The protocol can be implemented, and indeed is implemented by ourselves, without using any underlying reliable communication primitives. The protocol can, of course, be implemented using reliable communication primitives, however, there is really no need to do so. One can simply implement basic heartbeats, and then map the token part of the protocol on top of the heartbeats by including a sequence number and an acknowledge number. It is a trivial sliding window that needs to be implemented because it is data-less in nature. The result is that this protocol can be implemented with little more effort than a naïve heartbeat or ping solution with essentially no additional strain on the system.

6 Conclusions

Our main contributions are the creation of a a simple, *stable protocol* for monitoring connectivity that maintains a *consistent history* with *bounded slack*, and proofs that the protocol satisfies all these criteria. The protocol we've explained provides a mechanism for keeping the reporting of the channel state between two nodes consistent within a given slack. Consistency in the reporting of errors such as link connectivity problems can simplify the writing of applications acting on such error conditions, improving the overall reliability of a distributed system. A minimal slack of $N = 2$ is necessary for any protocol trying to guarantee consistency and still reflect the true state of the channel. A greater value for the slack let's the user of such a protocol tailor the degree to which the connectivity reporting truly reflects the current state of the channel at the expense of how tightly coupled the two nodes histories must be.

There exists more work to be done in monitoring network errors in a distributed system. One straightforward extension is the reporting of connectivity of a clique of nodes. Such a protocol would tell whether a group of nodes is fully connected or not, and make the same guarantees of consistent history as the link protocol. Another useful protocol would be one that reports whether a given node is isolated from the group. Then, if any single node ever sees itself as isolated, all other group members also see it as isolated. A common trait to all these problems is that the decision being made is a *binary* one, for only then can one part of a system make a decision about a change in state *without communicating with other components*. This is a necessary condition so that decisions can be made independently, and eventual consistent history can still be guaranteed.

A Appendix

As an implementor, here is the state machine to look at. The 5-state machine shown in Section 3.2 is good for arguing the slack, stability, and correctness. This equivalent 2-state machine parameterized by t is easier when implementing the protocol.

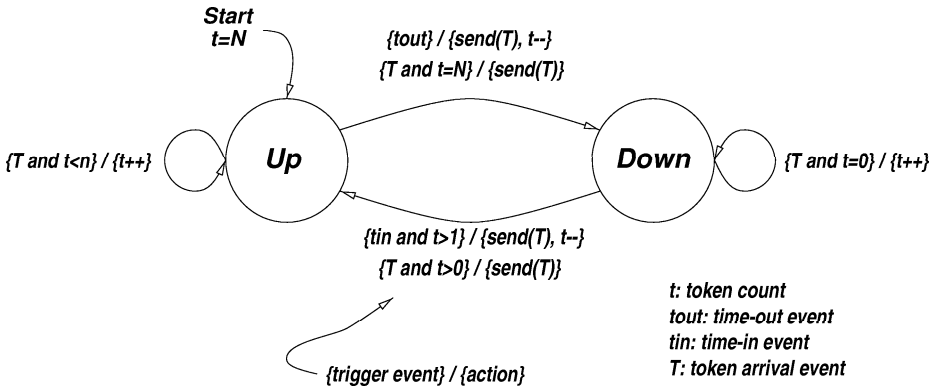

Figure 16: State machine for the connectivity protocol, general slack N. Each state is characterized by whether the node sees the channel as *Up* or *Down*, and how many tokens t are held by the node. The state transitions are labeled by the pair $\{$*action triggering the transition*$\}$ / $\{$*action taken upon transition*$\}$. A trigger event is either a timeout t_{out}, a time-in t_{in}, or receipt of a token T. The action taken is a combinations of sending tokens and adjusting the token count t. Multiple $\{$*event*$\}$/$\{$*action*$\}$ pairs may be specified for a transition, in which case if *event 1* was the trigger *action 1* is taken, or if *event 2* was the trigger *action 2* is taken.

References

[1] K. P. Birman, B. B. Glade, "Reliability Through Consistency," *IEEE Software*, vol. 12, no. 3, pp. 29-41, May 1995.

[2] K. P. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, 5(1), pp. 47-76, February 1987.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su, "Myrinet: A Gigabit per Second Local Area Network," *IEEE-Micro*, vol. 15, no.1, pp. 29-36, February 1995.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *Journal of the ACM*, 43(4), pp. 685-722, July 1996.

[5] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, 43(2), pp. 225-267, March 1996.

[6] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, 32(2), pp. 374-382, April 1985.

[7] G. J. Holzman, *Design and Validation of Computer Protocols*, Prentice Hall, New Jersey, 1991.

[8] N. Lynch, *Distributed Algorithms*, Morgan Kaufman, New Jersey, 1996.

[9] T. L. Rodeheffer and M. D. Schroeder, "Automatic Reconfiguration in Autonet," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Operating Systems Review 25(5), pp. 183-197, October 1991.

[10] T. L. Rodeheffer and M. D. Schroeder, S. Mullender ed., "A Case Study: Automatic Reconfiguration in Autonet," *Distributed Systems*, 2nd ed., ACM Press, New York, Chap. 11, pp. 283-313, 1993.